AD-A256 341

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		FINAL 1 Nov 89 - 30 Jun 92

4. TITLE AND SUBTITLE
"RESEARCH IN PROGRAMMING LANGUAGES & SOFTWARE ENGINEERING"(U)

5. FUNDING NUMBERS
61102F
2304/A2

6. AUTHOR(S)
Drs. Victor R. Basili, John D. Gannon, Marvin V. Zelkowitz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Sciences
College Park MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
AFOSR-TR-

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SPONSORING/MONITOR AGENCY REPORT NUMB
AFOSR-90-0031

13 pgs

409022

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
Distribution unlimited

12b. DISTRIBUTION CODE
UL

13. ABSTRACT *(Maximum 200 words)*

The following describes results from three major activities: a risk-based model of software decision making, construction of models for software development processes, and verification of safety properties of software requirements specification. The Risk-Based Model of Software Decision Making: The recognition that the "upstream" activities (requirements analysis, systems analysis and design) have a greater impact upon total life cycle costs and reliability than do "downstream" activities (coding and testing) has led to a greater emphasis on understanding, formalizing and developing these former activities. The ability to evaluate the implications of decisions in a project before coding permits easier and more cost effective control over the development process. Modeling Software Engineering Experience. They have been working on the notion of an improved approach to software development which reuses all forms of software knowledge to improve the software process and product. Based upon the Quality Improvement Paradigm (QIP), they have designed an evolutionary approach to software development that creates packages of experience for an organization that can be reused to improve their software production. Verifying Requirements Properties. A software requirements document is usually the first description of a system's required behavior. Errors in this document are difficult and expensive to correct if they are propagated to the design phase (or worse, to the implementation). While reviews by experienced systems designers are effective for improving the quality of requirements, significant improvements can be made if the automated analysis techniques can be applied to formal system descriptions.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	12
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

FINAL TECHNICAL REPORT

Research in Programming Languages and Software Engineering

AFOSR-90-0031
1 Nov 89 - 30 Jun 92

Principal Investigators
Victor R. Basili
John D. Gannon
Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park MD 20742

September 10, 1992

1 Introduction

This report summarizes the activities during the period May 1, 1991 through June 30, 1992. The following three sections describe results from three major activities: a risk-based model of software decision making, construction of models for software development processes, and verification of safety properties of software requirements specifications.

2 A Risk-Based Model of Software Decision Making

The recognition that the "upstream" activities (requirements analysis, systems analysis and design) have a greater impact upon total life cycle costs and reliability [27] than do "downstream" activities (coding and testing) [24] has led to a greater emphasis on understanding, formalizing and developing these former activities. The ability to evaluate the implications of decisions in a project before coding permits easier and more cost effective control over the development process.

Much current research centers on the development of formal methods [15] for software design. Such methods often focus on correct functionality of a product [17]. Although critically important, functionality is only one attribute needed to evaluate the effectiveness of a large system development [28]. One must also be concerned about scheduling, costs, reliability, performance, and numerous other attributes beyond those of simply correct behavior.

Completely formal system development, as an analog to precise mathematics, is unrealistic because of the truth that mathematics is also plagued with errors. As the famous mathematician Richard W. Hamming states [18]:

> "For generations, we have patched up the proofs of Euler, Gauss, and others less famous; and we expect future generations to patch up proofs for our theorems. The myth that there is absolute, ultimate rigor in mathematics goes on despite evidence to the contrary. Why do people believe that if we are completely rigorous enough, we can get completely reliable programs?"

In addition to the difficultly of producing correct formal descriptions for computer systems, software development is impacted by being a human activity. Therefore human attributes, as well as unforeseen environmental occurrences, increase uncertainty. Software project managers have preconceived ideas of how to proceed, and corporate policies affect decision making. Each project is developed under unique circumstances; thus, forecasting the impacts of different decisions is very difficult.

Prototyping has been used as a mechanism for evaluating decisions. The spiral model [6] is often cited as an alternative to the standard waterfall model as a framework of a life cycle model that emphasizes prototyping and risk analysis of the design process as basic concepts. However, the spiral model is an informal description of the development process. What is needed is a formal theory that fleshes these details. It is towards this end that this research is focused.

An appropriate formal model of software development must include certain probabilistic behavior to account for some of the uncertainty inherent in the process. We have developed a process model by adapting the functional correctness model [25] (i.e., related to denotational semantics and the Vienna Development Method (VDM)) with risk reduction features borrowed from economic decision theory.

Given alternative design strategies, we have been developing a theory of risk analysis that permits the software manager to evaluate different designs and determine the probable success or failure of each. It is realistic in that it permits each manager or organization to tailor the model to his own level of risk behavior. A model like this potentially can be used to formalize the spiral model and allow for transition of this method to new Department of Defense and other industrial software environments for improved management of the development process.

Developing the Model

Formal specifications for complex systems need to go beyond functionality. We must consider attributes like performance, schedule, resource usage along with functionality, in order to appropriately decide on the best development strategy. In addition, software development – like any other management approach – must weigh various decisions based upon unknown future events in order to decide on an appropriate course of action at the present.

This strongly suggests that techniques from decision theory, economic theory and various risk analysis models can play a role in software decision making. We have begun to develop such a theory. Our model has the following characteristics:

- We studied the role specifications play in the design of software and developed a model of the decisions that need to be made [26].
- We defined the concept of a *viable* specification as a multidimensional extension to the usual definition of functional correctness [25]. We defined an evaluation strategy based upon a metric we call the *performance level* in order to evaluate alternative designs to a give specification [10].
- We extended the model to handle risk [12]. That is, when there are alternative states of nature, with a given probability distribution, we use equilibrium probabilities to query the software manager in order to determine the performance level for taking each of the possible approaches. This model allows for variations and alternative strategies of different software managers. In particular it can be tailored to the risk-seeking and risk-adverse manager to create development strategies best suited for their individual tastes [11] [13].
- We built a prototype implementation called *Selector* using these ideas. The software manager provides the set of attributes and for each potential solution provides the ordinal ranking of how well that solution meets the specifications. *Selector* guides the manager into evaluating various risks and seeks to determine which form of prototype provides the most information [12].

The following briefly summarizes this development.

Assume that specifications for a program are vectors of attributes, including functionality as one of the elements of the vectors. Let X and Y be vectors of attributes that specify alternative solutions to a specification B. Let S be a vector of objective functions with domain being the set of specification attributes and range [0..1]. We call S_i a *scaling function* and it is the degree to which a given attribute meets its goal. We state that X *solves*$_S$ Y if $\forall i, S_i(X_i) \geq S_i(Y_i)$. Design x is viable (i.e., is correct) with respect to specification B and scaling function vector S if and only if P *solves*$_S$ B.

Each attribute may not have the same importance. Assume a vector of weights W called *constraints* such that each $w_i \in [0..1]$ and $\sum w_i = 1$. The *performance level* merges multiple scaled attributes and their constraints. Given specification vector X, scaling function S and constraints W, the performance level is given by: $PL(X, S, W) = \sum_i (w_i \times S_i(X_i))$.

Given a specification vector B, scaling vector S, constraints W and potential solutions x and y, X *improves* Y with respect to (B, S, W) if and only if:

1. X *solves*$_S$ B and Y *solves*$_S$ B
2. $PL(X, S, W) > PL(Y, S, W)$.

Since future behavior is not known during development, there is a degree of uncertainty to this process. We can use appropriate techniques from decision theory to evaluate potential solutions. We represent the performance level as a matrix PL where $PL_{i,j}$ is the performance level for solution i under state of nature j. As before, the performance levels give a measure of how good a system is. We can approximate this by defining the entries $PL_{i,j}$ of performance level matrix PL as the payoff (e.g., monetary value) for solution i under state j.

When the probability for each state of nature can be estimated, we can use expected values to achieve an estimated performance level. Given probability distribution vector P, where p_i is the probability that state of nature st_i is true, the expected payoff for alternative X^i is given by:

$$v_i = \sum_j pl_{i,j}\, p_j \tag{1}$$

Use the decision rule: Choose X^i which maximizes v_i or:

$$\max_i (\sum_j pl_{i,j}\, p_j) \tag{2}$$

Risk aversion plays an important role in decision making. This implies subjective behavior on the part of the software manager. We assume that the following *reasonable* behavior rule applies:

- *Decomposition:* Given three payoffs $a \leq b \leq c$, there exists a probability ρ such that the decision maker is indifferent to the choice of a guarantee of b, and the choice of getting c

with probability ρ and getting a with probability $1 - \rho$. We shall refer to this probability as $decomp(a, b, c)$.

Let pl_0 be the minimal value in our payoff PL and let pl^* be the maximal value. We decompose each $pl_{i,j}$ as $e_{i,j} = decomp(pl_0, pl_{i,j}, pl^*)$. This decomposition creates an equivalent pair of payoffs $\{pl_0, pl^*\}$, with probability $e_{i,j}$ of getting the more desirable pl^*. Any element $e_{i,j}$ will satisfy the following inequality:

$$pl_0 \times (1 - e_{i,j}) + pl^* \times e_{i,j} \geq pl_{i,j} \tag{3}$$

The difference between the two sides of this equation reflects the manager's degree of risk averseness. If the two sides are equal, risk analysis reduces to the expected value.

A prototype implementation of our evaluation strategy has been built in C. A manager enters a table of attributes and initial constraints and then executes *Selector*. The manager is prompted for the various equilibrium probabilities, which determine the risk averseness behavior of that particular individual as well as objective characteristics of the particular solution being considered. The tool then computes the performance level for each potential solution, computes the potential gain from prototyping and offers advice on which attribute would provide the maximum gain if it were investigated.

Validating the Model

Given the various unknowns in the states of nature, the software manager may choose to get more information with a prototype so that a better final decision can be made. However, before undertaking the procedure to extract more information, one should be sure that the gain due to the information will outweigh the cost of obtaining it. Here, we try to establish an absolute boundary: What is the value of perfect information?

If we know what is the true state of nature, we can choose the alternative that gives the highest performance level:

$$\Phi = \sum_j p_j \times \max_i pl_{i,j} \tag{4}$$

What is the value of this perfect information? The difference between our expected value and this improved performance level is the maximum our prototype can achieve, and still be cost effective.

We can now merge this into the definition of the spiral model. Assume we build a prototype to test which state of nature will be true. Given prototype P and specification B, P is a valid prototype if P *solves* B for a subset of the attributes of B. Depending upon the cost of P and the potential gain, we have the decision procedure that we need to make an effective spiral software process model.

3 Modeling Software Engineering Experience

We have been working on the notion of an improved approach to software development which reuses all forms of software knowledge to improve the software process and product. Based upon the Quality Improvement Paradigm (QIP), we have designed an evolutionary approach to software development that creates packages of experience for an organization that can be reused to improve their software production. The steps of the QIP are:

- Characterize the current project and its environment.
- Set the quantifiable goals for successful project performance and improvement.
- Choose the appropriate process model and supporting methods and tools for this project.
- Execute the processes, construct the products, collect and validate the prescribed data, and analyze it to provide real-time feedback for corrective action.
- Analyze the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.
- Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base for future projects.

The Experience Factory

The QIP requires the support of an organization that packages, stores and retrieves information. To support the QIP, the Experience Factory concept was created. The Experience Factory represents a form of laboratory environment for software development where models can be built and provide direct benefit to the projects. It represents an organizational structure that supports the QIP by providing support for learning through the accumulation of experience, the building of experience models in an experience base, and the use of this new knowledge and understanding in current and future project developments.

The Experience Factory concept supports a software evolution model that takes advantage of newly learned and packaged experiences; a set of processes for learning, packaging, and storing experience; and the integration of these two sets of functions. As such, it requires separate logical or physical organizations with different focuses/priorities, process models, and expertise requirements. It consists of a Project Organization whose focus is product delivery and an Experience Factory whose focus is to support project developments by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. The Experience Factory packages experience by building informal, formal or schematized, and "productized" models and measures of various software processes, products, and other forms of knowledge via people, documents, and automated support.

Our research paradigm has been focused on the storage, access and integration of them in a form of experience base and formalizing the development of data models.

TAME Model Building

Since one of the goals of the Experience Factory is to build models, we have developed methods to do better data modeling for the software engineering domain. In order to plan,control and evaluate the software development process, we need to collect data, analyze them and build accurate, easy to use and interpretable models. Building software engineering models is a difficult task. The data collected is very often noisy and heterogeneous, presenting many problems with respect to model construction (e.g. interdependencies, outliers, complex relationships). Measurement theory, statistics and machine learning provide many alternative modeling techniques and processes. However, mainly because of the lack of data, few experiments in our field have been performed in order to study, compare, validate and integrate the various approaches for standard software modeling issues (e.g. detecting high risk components).

Classical techniques (e.g., regression analysis) are not always well suited for software engineering data analysis. To build effective models in the Experience Factory, we developed a modeling approach for analyzing software engineering data, called Optimized Set Reduction (OSR), which addresses many of the problems associated with the usual approaches. OSR is based on both statistics and machine learning techniques. It generates a set of patterns relevant to predictions to be made about an object or to general trends in an entire data set.

The goal of the OSR algorithm is to determine which subsets of experiences (i.e. pattern vectors) from the historical data set provide the best characterizations of the object to be assessed. For example, assume we want to assess a particular characteristic of an object (e.g. the fault density of a component). We refer to this characteristic as the dependent variable (Y). We try to determine which subsets of the data set yield the "best" probability distributions on the Y range. A good probability distribution on the Y value domain is one that a concentrates a large number of pattern vectors in either a small part of the range (Y is continuous) or a small number of dependent variable categories (Y is discrete). One of the commonly used probability distribution evaluation functions is the information theory entropy H. Each of the subsets yielding "optimal" distributions, referred to as optimal subsets, are characterized by a set of conditions, or predicates which are true for all pattern vectors in that subset. Each set of predicates characterizing a subset is called a pattern.

Methods have been developed for using OSR for prediction, risk management and quality evaluation. An experiment demonstrating the effectiveness of the technique for software cost estimation is described in [7], which will appear in the *IEEE Transactions on Software Engineering* as part of the special issue on software modelling and measurement.

Based upon our need to develop and evaluate data analysis techniques for building software engineering models, we have also worked on the problem of detecting high risk software components using two modeling approaches based on totally different principles and theories. The first one is the logistic regression approach, a fairly standard technique for classification in the field of statistics.

The second one has been OSR. We have compared the results obtained by the two modeling approaches from several perspectives. We tried to determine how accurate they are in terms of classification correctness (i.e. high risk over low risk component) and completeness (i.e. percentage of high risk components detected). We also looked at understandability issues such as those listed below.

- What kind of insights does the model yield with respect to the studied software problem?
- How does the model help software engineers to refine their qualitative reasoning rules with respect to project management based on empirical evidence?

OSR has shown to be very effective for model building. The results of this

work [8] will be partially presented at the International

Conference on Software Reliability in October of this year. Another article is in preparation in order to complete and enhance the results presented at the conference.

4 Verifying Requirements Properties

A software requirements document is usually the first description of a system's required behavior. Errors in this document are difficult and expensive to correct if they are propagated to the design phase (or worse, to the implementation) [22]. While reviews by experienced systems designers are effective for improving the quality of requirements, significant improvements can be made if automated analysis techniques can be applied to formal system descriptions.

SCR Requirements Specification

Software Cost Reduction (SCR) requirements specifications [1, 19] describe a system's modes of operation and events that cause the system to change modes. SCR-style requirements specifications model a system's behavior as a set of finite-mode machines that execute concurrently. The basic concepts in these requirements are modes, mode classes, and events.

- A *mode* is a set of system states that share a common property.
- A *mode class* is a set of modes, and the union of the modes in a mode class must cover the system's state space.
- A *mode transition* occurs between modes in the same mode class as a result of system state changes.
- Mode transitions are specified by *conditions* and *events*, which comprise the machine's input language.

The system is in exactly one mode of each mode class at all times. Informally, the modes and transitions in each mode class form a mode-machine that describes one aspect of a system's behavior, and the transitional behavior of the entire system is defined by the composition of all the system's mode-machines.

Conditions are boolean state variables, and a system's state space is the set of all possible combinations of its conditions' values. Partitioning the state space into modes reduces the size of requirements. A mode transition is activated by the occurrence of an event, which represents the point in time when a condition's value changes. For example, event

@T(*Cond1*) WHEN [*Cond2*]

occurs when condition *Cond1* becomes true while condition *Cond2* is true. More complex events can be created from simpler events and conditions using boolean operators.

A mode transition condition specifies the event that triggers the transition. Two transitions from the same mode are simultaneously enabled if their trigger events occur at the same instant. In such a case, the mode machine is nondeterministic, and the activation of either transition (but not both) satisfies the requirements.

The tabular format of SCR-style requirements specification is both easy to write and easy to understand. Even so, a requirements designer will often augment the behavioral specification with a set of global constraints on the system's behavior. The purpose of the global constraints is to give a compact view of a system's invariant properties that may otherwise be difficult to extract from a behavioral specification. Sometimes, the global constraints are explicitly included in the requirements document, though they may not be expressed mathematically; their format can range from logical formulas [20] to natural language sentences [1]. Other times, they are not included in the requirements document, but are implicit assumptions made by the requirements designer [23].

Model Checking

If a system's behavioral requirements can be represented as a set of communicating finite-state machines, Clarke *et. al.* [14] have devised a model checking algorithm to determine which states of the system satisfy temporal logic assertions. We use an improved version of Clarke's original model checking system, called MCB [9], as our model checker. Formulas to be checked are expressed in a propositional branching-time logic called computational tree logic (CTL), whose operators permit explicit quantification over all possible futures.

Most elements of the SCR requirements model correspond naturally to elements of finite-state machines. However, there is no natural model of events because of the differences between mode transitions and state transitions. State transitions occur based on the current state and the current values of the input conditions. Mode transitions occur simultaneously with events; the system spends zero time in a mode once one of its transitions has been activated. Therefore, we need to be able to represent changes in condition values and ensure that state transitions are activated by these value changes.

To model events, we can represent modes as two states: a *mode state* and a *trigger state.* The mode state represents the system when it is in a mode; all transitions into the original system mode are modeled as transitions into the representative mode state. The trigger state represents the system as it leaves a mode; all transitions leaving the original mode are represented as transitions leaving the associated trigger state. We annotate the trigger state's transitions with the original mode's transition conditions, and we annotate the transition from the mode state into the trigger state with the negation of the transitions' trigger events.

We used our analysis technique to analyze two requirements documents, one for an automobile cruise control system and one for a water-level monitoring system [3]. These requirements were written by other groups of researchers. We transformed these systems' requirements into finite-state machines, rephrased required safety properties as logical formulas, and verified the formulas using the MCB model checker. In both studies, we found discrepancies between the systems' requirements specifications and their safety assertions.

While this technique is quite promising, only the model checking portion is currently automated. The translation of SCR requirements to finite-state models is still an error-prone manual process. This proposal will fund work to automate this translation, extend the safety properties that can be verified to include assertions about time bounds, and provide a way to do static analysis of implementations to determine if they are consistent with requirements.

References 2–5, 7–8, 10–13, 16–17, 21, and 24–28 report results of our AFOSR-sponsored research.

References

[1] Alspaugh, T., S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore, "Software requirements for the A-7E aircraft," Naval Research Laboratory, Washington, D. C., (March 1988).

[2] Antoy S., P. Forcheri, M. T. Molfino, and M. Zelkowitz, "Rapid prototyping of system enhancements," 1st IEEE Int. Conf. on System Integration, Morristown, NJ (April, 1990), 330-336.

[3] Atlee J.M. and J.D. Gannon. "State-based model checking of event-driven system requirements." Proceedings of SIGSOFT'91 Software for Critical Systems, (December 1991). An updated version of this paper will appear in *IEEE Trans. on Soft. Eng.* (January 1993).

[4] Basili V. R., "Software development: A paradigm for the future," *Proc. IEEE Compsac 89*, pp. 471-485.

[5] Bennet, T, "A Pragmatic Set Operation and its Implementation in C." Proceedings of the '92 Symposium on Applied Computing, (1992).

[6] Boehm B., "A spiral model of software development and enhancement," *IEEE Software* (5)3:61-72 (1988).

[7] Briand, L., V.R. Basili and C. Hetmanski, "Providing an empirical basis for optimizing the verification and testing phases of software development," IEEE International Symposium on Software Reliability Engineering, North Carolina, (October 1992).

[8] Briand, L.C. V.R. Basili and W.M. Thomas, "A pattern recognition approach for software engineering data analysis," *IEEE Trans. on Soft. Eng.*, (November 1992), (to appear).

[9] Browne, M., Automatic verification of finite state machines using temporal logic," Carnegie Mellon University, Computer Science Department, Ph.D. Thesis, (1989).

[10] Cárdenas S. and M. V. Zelkowitz, "Evaluation criteria for functional specifications," ACM/IEEE 12th Int. Conf. on Soft. Eng., Nice Fr (March, 1990), 26-33.

[11] Cárdenas-García S., "A formal framework for evaluation of multiattribute specifications," PhD dissertation, Department of Computer Science, University of Maryland, June, 1991.

[12] Cárdenas-García S., and M. V. Zelkowitz, "A management tool for evaluation of software designs," *IEEE Trans. on Software Engineering* 17, 9 (1991) 961-971.

[13] Cárdenas S., J. Tian and M. V. Zelkowitz, "An application of decision theory for the evaluation of software prototypes," *Journal of Systems and Software* (12) (1992).

[14] Clarke, E., E. Emerson and A. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications," *ACM Trans. on Prog. Lang. and Sys.* 8, 2, (April, 1986), 244-263.

[15] Craigen D. (Ed.), *Formal Methods for Trustworthy Computer Systems FM89*, Springer Verlag, (1990).

[16] Cui, Q. and J. Gannon, "Data-oriented exception handling." *IEEE Trans. Soft. Eng.*, (May 1992), 393-401.

[17] Gannon, J.D., J.M. Purtilo and M.V. Zelkowitz, *A comparison of verification methods* Ablex Publishing Co., Norwood, NJ (1993) *to appear.*

[18] Hamming R. W., "Review of 'Mathematical foundations of computer science, Volume I' ", *IEEE Computer* 25, 1 (1992) 134-135.

[19] Heninger, K. "Specifying software requirements for complex systems: New techniques and their applications," *IEEE Trans. Soft. Eng.*, SE-6, 1, (January, 1980), 2-12.

[20] Kirby, J. "Example NRL/SCR software requirements for an automobile cruise control and monitoring system," Wang Institute of Graduate Studies, TR-87-07, (July, 1987).

[21] Oivo, M., and V.R. Basili, "Representing software engineering models: The TAME goal oriented approach," *IEEE Trans. on Soft. Eng.* (October 1992), (to appear).

[22] Parnas, D. and J. Madey, "Functional documentation for computer systems engineering," Queen's University, Department of Computer Science, Kingston, Ontario, TR-90-287, (September, 1990).

[23] van Schouwen, J. "The A-7 requirements model: re-examination for real-time systems and an application to monitoring systems," Queen's University, Department of Computer Science, Kingston, Ontario, TR 90-276, (May, 1990).

[24] Zelkowitz M.V. "Evolution towards a specifications environment: Experiences with syntax editors," *Information and Software Technology* 32, 3 (April, 1990) 191-198.

[25] Zelkowitz M. V., "A functional correctness model of program verification," *IEEE Computer* 23, 11 (November, 1990) 30-39.

[26] Zelkowitz M. V., S. Cárdenas and P. Straub, "Evaluation of software design quality," 3rd International Workshop on Software Quality Improvement, Tokyo, Japan (January, 1991).

[27] Zelkowitz M. V. and S. Cárdenas, "The role for executable specifications in system maintenance," *Information Sciences Journal*, 57 (1991) 347-359.

[28] Zelkowitz M. V., "The role of verification in the software design process," *Advances in Computers* (38) (1993) *(to appear).*